

Failure Detection and Repair of Threads in CTAS

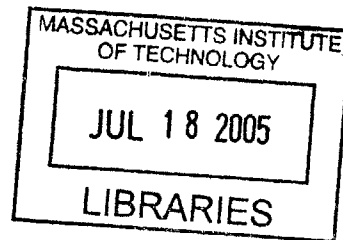
by
Farid Jahanmir

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

August 17, 2004

© 2004 Farid Jahanmir. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.



Author _____

Department of Electrical Engineering and Computer Science
August 17, 2004

Certified by _____

Martin C. Rinard
Thesis Supervisor

Accepted by _____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

BARKER

Failure Detection and Recovery of Threads in CTAS

by
Farid Jahanmir

Submitted to the
Department of Electrical Engineering and Computer Science

August 17, 2004

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Reliable, error-free software is hard to come by, and this is especially true for newer, larger, or more complex programs. CTAS, an air traffic control tool, falls into this category, making it a good candidate for research on error compensation. Specifically, this thesis addresses the issue of thread crashes in one portion of CTAS. We reimplement the thread structure in question around a simpler problem, and develop a failure detector and an accompanying repair mechanism to monitor it. These add-on components provide the application with thread consistency by swiftly and transparently recovering from crashes, thereby yielding a more stable, self-sufficient, and generally more reliable operating environment.

Thesis Supervisor: Martin C. Rinard
Title: Associate Professor

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Martin Rinard, for giving me the opportunity to do my thesis under him, and for his guidance, support, and encouragement in conducting this research. I would also like to thank Daniel Roy for accompanying me to NASA Ames Research Center to learn about CTAS first-hand, and Brian Demsky for helping me revise my thesis proposal and for guidance regarding CTAS. Michelle Eshow and John E. Robinson of NASA Ames Research provided detailed explanations of how CTAS and its PFS_E module work. Lastly, I would not have been able to complete this thesis without the unflinching support and encouragement of my friends and my family. To them I am eternally grateful.

Table of Contents

ABSTRACT 3

Acknowledgements..... 5

Table of Contents 7

List of Figures..... 11

List of Tables 13

Chapter 1: Introduction 15

Chapter 2: Failure Detectors 17

 2.1 Theoretical Foundations 17

 2.2 Quality of Service..... 18

 2.3 Monitoring Strategies 20

 2.3.1 Heartbeats 20

 2.3.2 Pings 21

 2.3.3 Comparison..... 22

Chapter 3: CTAS	25
3.1 Terminology	25
3.2 Basic Operation	26
3.3 PFS_E Process.....	27
3.3.1 Thread Structure	27
3.3.2 Data Flow	29
3.3.3 Control Flow.....	29
 Chapter 4: System Design	 31
4.1 Application Module.....	31
4.2 Failure Detection Module.....	34
4.2.1 Freshness Points	34
4.2.2 Justification.....	37
4.3 Recover/Repair Module	38
 Chapter 5: Implementation.....	 39
5.1 Modules vs. Threads	39
5.2 Application Threads	41
5.2.1 Data Management.....	41
5.2.2 Thread Management.....	43
5.2.3 Manager/Worker Interface	45
5.2.4 Prime Discovery	47
5.3 Failure Detection Thread.....	49
5.3.1 Data Management.....	50

5.3.2 Communications.....	51
5.3.3 Timing & Monitoring.....	52
5.4 Recover/Repair Thread	55
Chapter 6: Performance.....	57
6.1 Failure Detection Module.....	57
6.1.1 Estimated Arrival Times.....	58
6.1.2 Safety Margins.....	60
6.2 Recover/Repair Module	65
6.3 Overhead	66
Chapter 7: Related Work.....	67
7.1 An Adaptable Failure Detector	67
7.2 ϕ -Failure Detector	68
7.3 Lazy Failure Detection	69
Chapter 8: Conclusion	71
Bibliography	73

List of Figures

Figure 1. Graphical Representation of QoS Metrics.....	19
Figure 2. Failure Detection Using Heartbeats.....	21
Figure 3. Failure Detection Using Pings.....	22
Figure 4. Non-Distributed Failure Detectors, Heartbeats vs. Pings.....	23
Figure 5. Time-Dependency of Heartbeats.....	24
Figure 6. PFS_E Process Thread Structure and Supporting Data Objects.	28
Figure 7. Application Work Flow.....	32
Figure 8. Application Module Thread Structure and Supporting Data Objects.	33
Figure 9. Failure Detection Using Pings and Freshness Points.	35
Figure 10. Breakdown of System Modules and Threads.....	40
Figure 11. Data Structures Used in the Application.	42
Figure 12. Finite State Machine Representing Possible Thread States and Transitions..	50
Figure 13. Communication Among Threads via Signals.....	52
Figure 14. Observed Communication Delays.....	54
Figure 15. Graph of n 's Effect on Estimated Arrival Time.	59
Figure 16. Graph of ϕ 's Effect on Maximum Detection Time.	61
Figure 17. Graph of γ 's Effect on Maximum Detection Time.....	62
Figure 18. Graph of Maximum Detection Time with Optimized Settings.	64

List of Tables

Table 1. Effect of ϕ on Maximum Detection Time and Average Mistake Rate.....	63
Table 2. Effect of γ on Maximum Detection Time and Average Mistake Rate.....	63
Table 3. Effect of FDM and RPM on CPU Times.....	66

Chapter 1: Introduction

Producing a reliable software system is traditionally done by performing rigorous testing to find and eliminate any bugs. These coding errors tend to corrupt data, often leading to undesirable program behavior or component failures. Unexpected inputs, concurrency issues, file system failures, and hardware errors can also cause software failures. Finding and addressing all these possible sources of failure can be extremely difficult for any software. For large, complex, and long-running systems with many modules interacting in different fashions, error elimination becomes virtually impossible. In this vein, error compensation can be a more attractive means for improving system reliability.

The Center-TRACON Automation System, or CTAS, is an example of a large, complex, and long-running system. It is a safety-critical system, in that its function is vital to managing air traffic. Rebooting CTAS to bring the system to a pre-crash equivalent state requires a significant amount of time, making crash-prevention extremely important. Furthermore, the Profile Selector Enroute (PFS_E) module needs to be highly responsive, as air traffic controllers use it to rapidly examine alternate flight plans. For these reasons, this module is a good candidate for research on error compensation.

We seek to enhance the reliability of the PFS_E module of CTAS in this thesis by gracefully detecting and recovering from thread failures. A failure detector designed to incur minimal overhead while interfacing seamlessly with the module will allow for the quick detection of crashed threads. A third component, a recover/repair module, will be notified of failed threads by the detector and appropriately repair them. These two add-

on components will allow the system to recover from failed threads swiftly and transparently, thereby allowing continued execution at full capacity without human intervention.

This thesis is broken down into 8 chapters:

Chapter 1 introduces the problem being investigated, how it will be approached, and the goals of the research.

Chapter 2 presents basic information regarding failure detectors, discussing including their theoretical foundations and general anatomy.

Chapter 3 provides an introduction to CTAS, relevant terminology, a general description of how the system functions, and detailed information about the thread structure, data flow, and control flow of the PFS_E module.

Chapter 4 describes the high-level design of the system this thesis seeks to build, providing the essential information about each of the three components.

Chapter 5 expands on the design from the previous chapter, giving a complete description of the system's resource management routines, communications mechanisms, and main algorithms.

Chapter 6 studies the performance characteristics of the system, evaluating how effectively it meets our goals.

Chapter 7 looks at other research regarding failure detectors.

Chapter 8 concludes this thesis with a discussion of what was accomplished, how it compares to related work, and possible directions for future work.

Chapter 2: Failure Detectors

Failure detectors periodically gather activity data for specified processes in a system, allowing them to detect possible crashes. The system is given access to a list of suspected failures maintained by the detector, and can choose to safely dispose of any associated structures or repair them. In this way, the system can ensure its components are executing properly and remain responsive, thereby improving system reliability.

In this chapter, we introduce the theory behind unreliable failure detectors and ways to classify them. Methods for gauging the quality of service provided by the detectors are also discussed, as are the two main strategies for monitoring processes.

2.1 Theoretical Foundations

Chandra and Toueg devised their model for *unreliable failure detectors* to solve the Consensus problem¹ for distributed systems [2]. They are *unreliable* because they are allowed to make mistakes – at any given time, a process could erroneously be *suspected* to have failed, called a *false positive*, or incorrectly *trusted* as being alive, called a *false negative*. Since failures are only suspected, the failure detector can also correct its mistakes. To outfit a system with such a failure detector, a local detector module is attached to each process. These local modules monitor some subset of the system's

¹ The Consensus problem involves having a set of processes come to a consensus on some decision even with failures.

processes and maintain separate suspect lists. A majority of the local modules must suspect a particular process for it to be globally suspected.

Unreliable failure detectors are distinguished by two properties – *completeness* and *accuracy*. *Completeness* defines how effectively a failure detector recognizes failed processes – “completeness requires that a failure detector eventually suspects every process that actually crashes” [2]. *Strong completeness* requires *all* local modules to suspect every crashed process, while only *some* need to for *weak completeness*.

Accuracy deals with the errors a detector can make. A failure detector that satisfies *strong accuracy* never makes a mistake for *any* process, and one providing *weak accuracy* never makes a mistake for *some* processes. *Eventual strong accuracy* and *eventual weak accuracy* allow mistakes for *any* process or *some* processes, respectively, as long as eventually they are correct. For example, for a detector to exhibit eventual strong accuracy, “there [must be] a time after which correct processes are not suspected by any correct process” [2].

2.2 Quality of Service

Expanding on previous work, Chen, Toueg, and Aguilera develop a measure of the performance of failure detectors [3]. Their specification, termed *quality of service* (QoS), focuses on the speed and accuracy of failure detection for probabilistic systems. It consists of three primary metrics – one for speed and two for accuracy:

detection time, T_D – the time from when a process crashes to when the failure is detected (see Figure 1).

mistake recurrence time, T_{MR} – the time from when a process is incorrectly suspected to when the same process is again incorrectly suspected (see Figure 1).

mistake duration, T_M – the time from when a process is incorrectly suspected as failed to when the process is again trusted (see Figure 1).

In addition, there are four derived metrics for accuracy:

average mistake rate, λ_M – the average number of mistakes over a given period of time.

query accuracy probability, P_A – the probability that at any time the failure detector's response to a query is accurate.

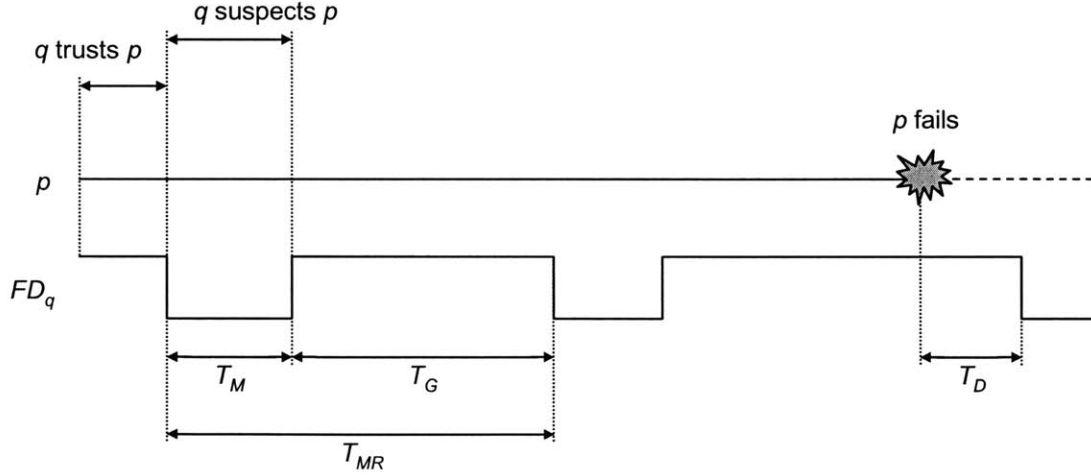


Figure 1. Graphical Representation of QoS Metrics. For process p and process q 's failure detector, FD_q , the graph shows how to measure T_D , T_{MR} , T_M , and T_G .

good period duration, T_G – the time from the last corrected false positive to the next detected false positive (see Figure 1).

forward good period duration, T_{FG} – the time from a random point in a good period to the next false positive.

2.3 Monitoring Strategies

In practice, two main strategies exist for determining a process's activity: using heartbeats and by pinging. Both involve periodically receiving short messages from the monitored processes and using time-outs to detect lateness and possible failures. The heartbeat method, however, is an interrupt-driven mechanism, while pinging requires the failure detector to poll monitored processes. A more detailed description of each method is presented in the following subsections.

2.3.1 Heartbeats

With the heartbeat strategy, the failure detector module for a process periodically sends a short message to all processes monitoring it. In other words, each process in the system is responsible for notifying interested processes of its existence. Hence, the messages are called *heartbeats*. For a process p monitoring a process q , q sends a heartbeat to p every Δ_i seconds, the *heartbeat interval*. Process p begins suspecting that q has failed if the time-out, Δ_{to} , expires before q 's next heartbeat arrives. Upon receipt of the heartbeat,

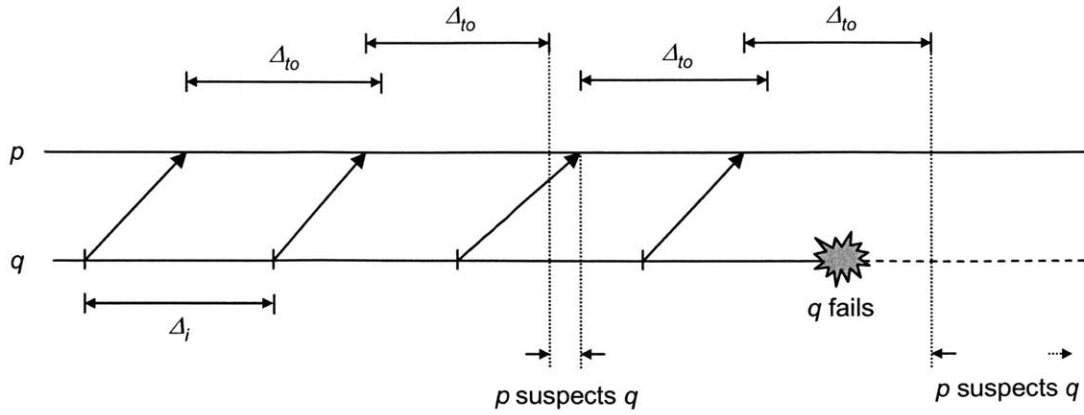


Figure 2. Failure Detection Using Heartbeats. Process p is monitoring process q , which sends a heartbeat every Δ_i seconds. p allows a maximum time of Δ_{to} to elapse after the receipt of the previous heartbeat before suspecting q .

process p stops suspecting q and starts the time-out for the next heartbeat. Figure 2 presents how the strategy works over time.

2.3.2 Pings

Each process's failure detector module periodically queries the processes it is responsible for in the pinging method. Once received by the monitored processes, these queries, or *pings*, are processed and replies are sent back. For a process p monitoring a process q , p pings q every Δ_i seconds, the *polling interval*. If Δ_{to} seconds elapses from the time the ping was sent and no reply has been received, process p begins suspecting that q has failed. Upon receipt of the next reply, process p stops suspecting q . Refer to Figure 3 for a diagram of the process.

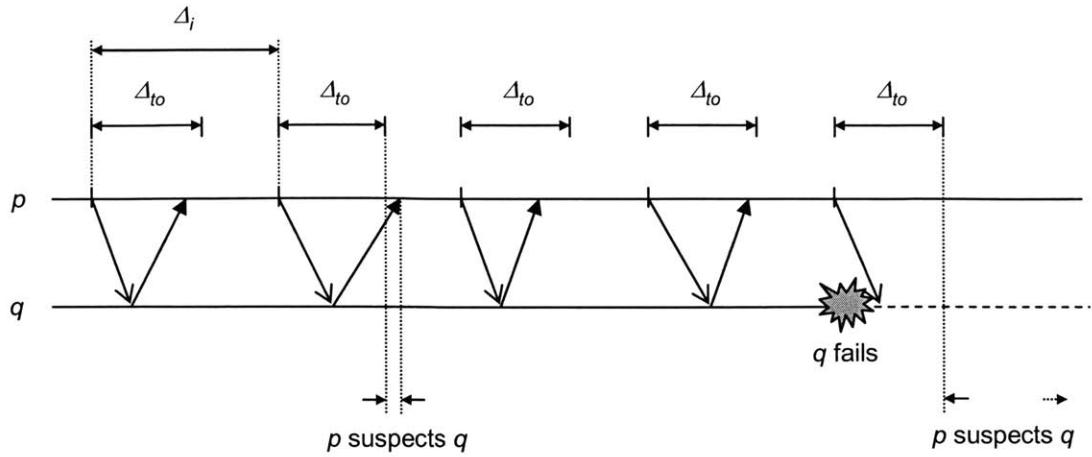


Figure 3. Failure Detection Using Pings. Process p is monitoring process q , and sends a ping every Δ_i seconds. p allows a maximum time of Δ_{to} to elapse after sending a ping before suspecting q .

2.3.3 Comparison

The heartbeat strategy and the pinging strategy are closely related, but the burden of timing is switched. Instead of the *monitored* processes being responsible for sending messages periodically, the pinging method has the *monitoring* process handle the timing. This shift makes a non-distributed pinging-based failure detector more centralized, as depicted in Figure 4 – only the process with the failure detector needs a timer, as opposed to all monitored processes.

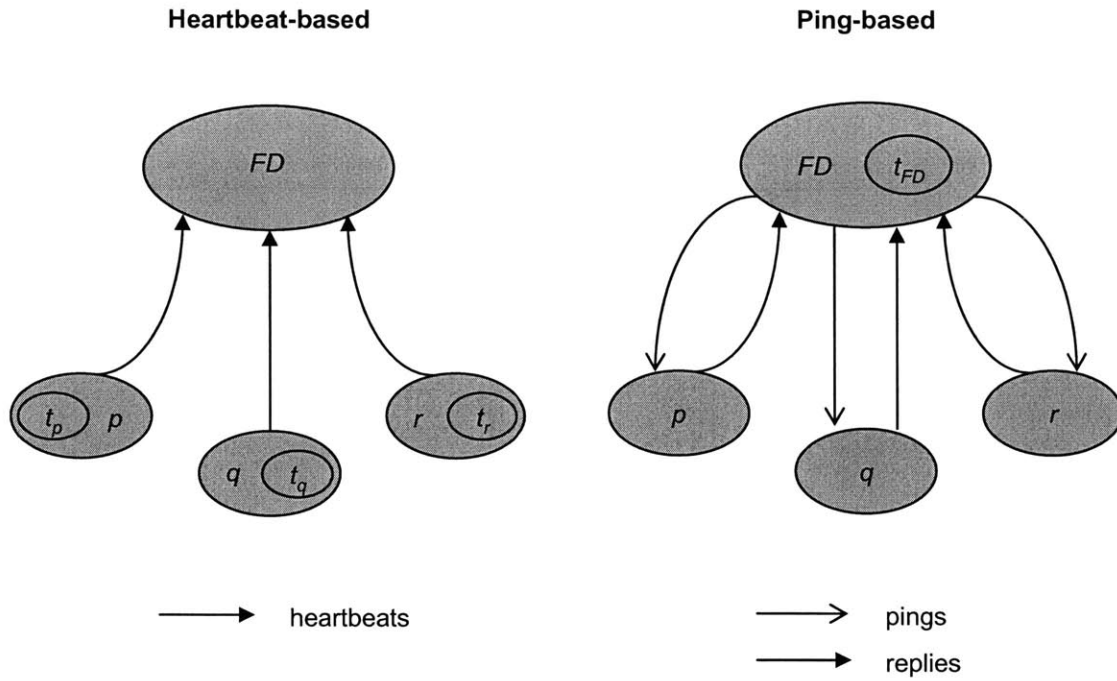
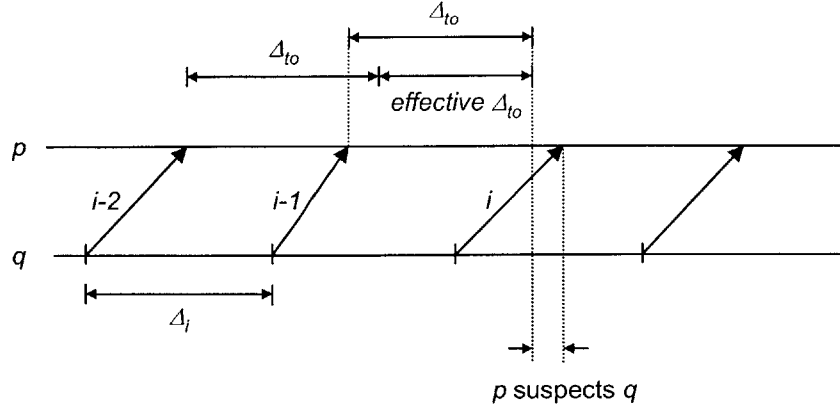


Figure 4. Non-Distributed Failure Detectors, Heartbeats vs. Pings. FD refers to the process with the failure detector, which is monitoring processes p , q , and r . t_i represents process i 's timer.

Any ping-based failure detector requires an additional message to be sent for every interval Δ_i , resulting in twice as many transmitted messages than a heartbeat-based detector. Since there are two message transmissions, each of which has a variable delay, the time between when pings are sent and when replies are received has a larger variance. This can be accounted for by using either a large time-out or one that adapts to changing conditions. Unfortunately, the former scheme hampers the detector's ability to detect failures quickly, and the latter can be difficult to implement effectively, often causing false positives. As a consequence, pinging tends to be less accurate than sending heartbeats.

(a)



(b)

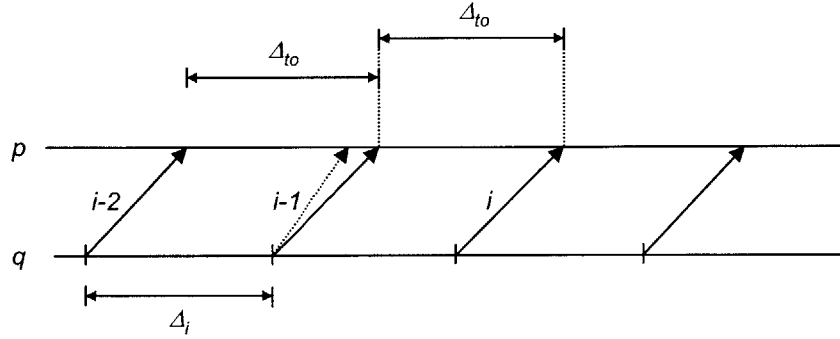


Figure 5. Time-Dependency of Heartbeats. (a) Process q 's $(i-1)^{th}$ heartbeat arrives early, causing the i^{th} heartbeat to be perceived as late. (b) Process q 's $(i-1)^{th}$ heartbeat arrives on time, and the i^{th} heartbeat is also perceived as on time.

For heartbeat-style detectors, detection time depends on the previous messages. Looking at Figure 5, the time-out for the i^{th} heartbeat doesn't start until the $(i-1)^{th}$ heartbeat has arrived. This behavior, which isn't exhibited by ping-based detectors, can result in premature time-outs and more false positives – if the $(i-1)^{th}$ heartbeat arrives early, the i^{th} heartbeat will effectively have a smaller time-out.

Chapter 3: CTAS

The Center-TRACON Automation System, or CTAS [1], is a large, complex suite of tools implemented via connected processes. Developed by engineers at NASA's Ames Research Center in Mountain View, California, the system provides a set of tools to air traffic controllers to aid in managing air traffic. Among the tasks of these tools is finding shorter flight plans, identifying potential conflicts between aircraft, and assisting in spacing the final approaches of aircraft to their runways. By automating a large portion of air traffic management, CTAS aims to reduce delays while increasing safety.

In this chapter, basic terminology regarding CTAS and air traffic control is presented, followed by an overview of how the system functions. Details regarding the Profile Selector En-route (PFS_E) process, including thread structure, data flow, and control flow, are also given.

3.1 Terminology

Some important terms related to CTAS and their definitions are:

Center – a geographic region surrounding a major airport

Aircraft track data – an aircraft's current 3D coordinates, heading, and velocity

Aircraft flight plan – a sequence of waypoints for the aircraft to fly over

Waypoints – a location on the ground marked by a radio beacon

Trajectory – a 4D vector consisting of x, y, altitude, and time coordinates, specifying the future flight path of an aircraft

ETA – estimated time of arrival; the time of the last point in a trajectory

Airspace definition – defines components of the airspace for the given center, including waypoints, airways, boundaries, airport and runway locations, and expected aircraft types

Conflict – whenever aircraft pass too close to each other

Direct-to routing – a suggested change to a flight plan that skips intermediate waypoints and goes directly to a later one, thereby saving flight time

3.2 Basic Operation

There is one CTAS system responsible for each center², and each center has a radar installation which gathers aircraft track data every 12 seconds. This track data, along with flight plans, weather conditions, and the airspace definition, is input to CTAS. The Trajectory Synthesizer (TS) process calculates trajectories for each aircraft, which are displayed in the GUI. Runway assignments, possible conflicts, and direct-to routings are generated with the trajectories. These resultant data are converted into advisories for each aircraft and are outputted to the controller. With the help of the generated advisories, controllers manage air traffic to ensure safety and punctuality.

² CTAS is currently deployed at seven of the 21 centers in the United States.

3.3 PFS_E Process

The PFS_E process of CTAS, recently re-implemented in C++ to be multithreaded, gives controllers the ability to trial-plan different flight plans for aircraft. With altered flight plans, an aircraft can save time by skipping intermediate waypoints or avoid conflicts by being rerouted through alternate waypoints. This process needs to be highly responsive from the controller's perspective, as he is rapidly changing the flight plan via the GUI; he needs to see how his changes affect the aircraft's trajectory and determine if the new course resolves the issue at hand.

3.3.1 Thread Structure

A top-level manager (MGR) thread oversees the other threads, data management, and work flow in the PFS_E process. Three other types of threads exist – some number of trial planning (TPX) threads, a 12-second update (TSU) thread, and a flight plan amendments (FPA) thread. TSU threads, TPX threads, and FPA threads are collectively referred to as worker (WKR) threads, and are allocated as needed from a pool of idle threads by the manager thread. Figure 6 depicts the thread structure and supporting data objects (see Section 3.3.2).

TPX threads are invoked on demand via the GUI, and each deals with exactly one aircraft. The thread receives an alternate, trial flight plan input by the controller and dispatches the data to the TS module for the calculation of 4D trajectories. These trajectories are received and sent to the GUI for display, and the user can further adjust the trial plan until it is satisfactory.

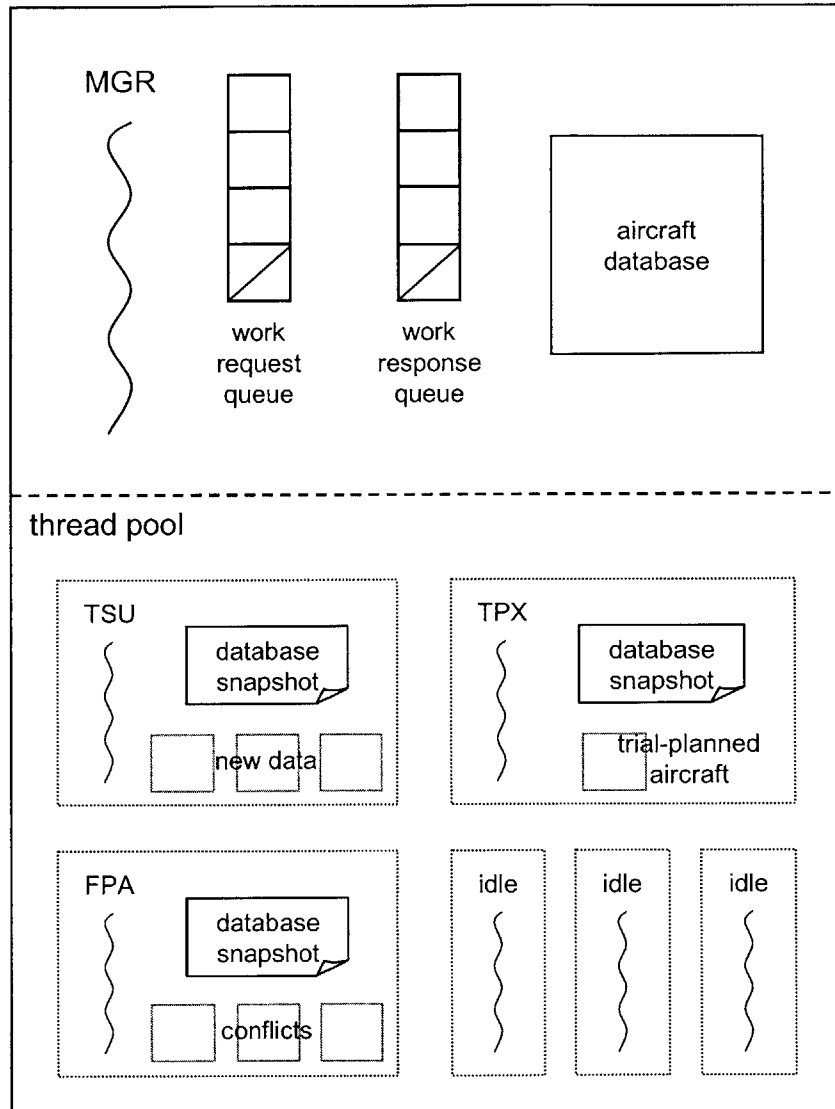


Figure 6. PFS_E Process Thread Structure and Supporting Data Objects.

Every 12 seconds, the TSU thread receives new flight plan data from radar feeds. New 4D trajectories are computed for all flights using this data. Once the computations are completed and the data is updated, the thread notifies the manager and waits for the next 12-second update.

FPA threads do the actual work of checking if trial-planned flight plans have added new conflicts to the system or circumvented existing ones. They are invoked in response to a TSU thread's update.

3.3.2 Data Flow

As shown in Figure 6, two queues are maintained by the manager thread: a work request queue and a work response queue. Work requests from other parts of CTAS are added to the queue as they are received. The manager thread assigns idle worker threads work requests from this queue and supplies them with the necessary data. Specifically, the manager thread maintains a master database of aircraft tracks and provides snapshots of it as needed; space for a worker's result is included. When a worker thread finishes processing the request, it writes the result into this space and notifies the manager. The worker's result is placed in the work response queue and is eventually sent to the requesting process. At this point, the manager deallocates all associated data structures.

3.3.3 Control Flow

In the absence of any external events, flow proceeds as follows:

- controller chooses an aircraft to trial-plan via GUI
- controller makes changes to flight plan via GUI
- changed flight plan submitted to MGR
- MGR assigns an idle TPX thread to work on the changed flight plan
- TPX has a TS module compute trajectories

- TPX returns new trajectories to MGR
- MGR gives trajectories to the GUI
- controller sees new trajectories in response to flight plan changes
- controller determines if new trajectories are satisfactory
- controller accepts the trial-planned changes
- aircraft receives changes in flight plan via a CTAS advisory
- FAA registers the new flight plan for the aircraft
- TSU receives the new flight plan on the next 12-second update
- FPA thread invoked to check for new conflicts
- MGR updates the master database to reflect the changes

In this fashion, the user can try several different flight plans for an aircraft to find one that avoids impending conflicts or hazards, or results in a shorter flight time.

Chapter 4: System Design

In this chapter, the designs for the three system components are presented – the reimplementation of the PFS_E module, the failure detection module, and the recover/repair module. The failure detection module and recover/repair module have been designed to incur minimal overhead and to interface seamlessly with the application. With the help of these add-on modules, the application will function efficiently and be able to cleanly and transparently recover from thread failures.

4.1 Application Module

Instead of using the PFS_E module in CTAS directly, the structure (see Section 3.3) has been reimplemented around a simpler problem and to be a stand-alone application. This simplification allows more effective testing and design – the entire CTAS system doesn’t need to be running and passing requests from a user to PFS_E, and aircraft data and trajectory calculations are not needed.

The reimplementation concerns discovering prime numbers. A candidate number is given to an idle TPX thread by the manager thread, and the TPX thread checks that number for primality. Discovered primes are sent to the TSU thread, which updates the list of discovered primes every 12 seconds. When the list is updated, the FPA thread checks its contents for correctness; it ensures all numbers in the list are indeed prime and

removes duplicates as a side effect. All threads then replace their snapshots with copies of this verified, updated primes list. Figure 7 depicts this work flow graphically.

As in the PFS_E module, the manager thread manages the workers and data, TPX threads do the actual work, the TSU thread updates the database every 12 seconds, and the FPA thread checks for conflicts. Each candidate number checked by a TPX thread is analogous to a trial-planned aircraft, a prime number to an accepted trial-planning, and the list of primes to the aircraft tracks database. To distinguish manager-related structures from worker-related ones, the application module is divided into two sub-modules – the manager sub-module (MSM), and the worker sub-module (WSM). Our modified application structure is presented in Figure 8.

External work requests and responses are removed, and internally they are

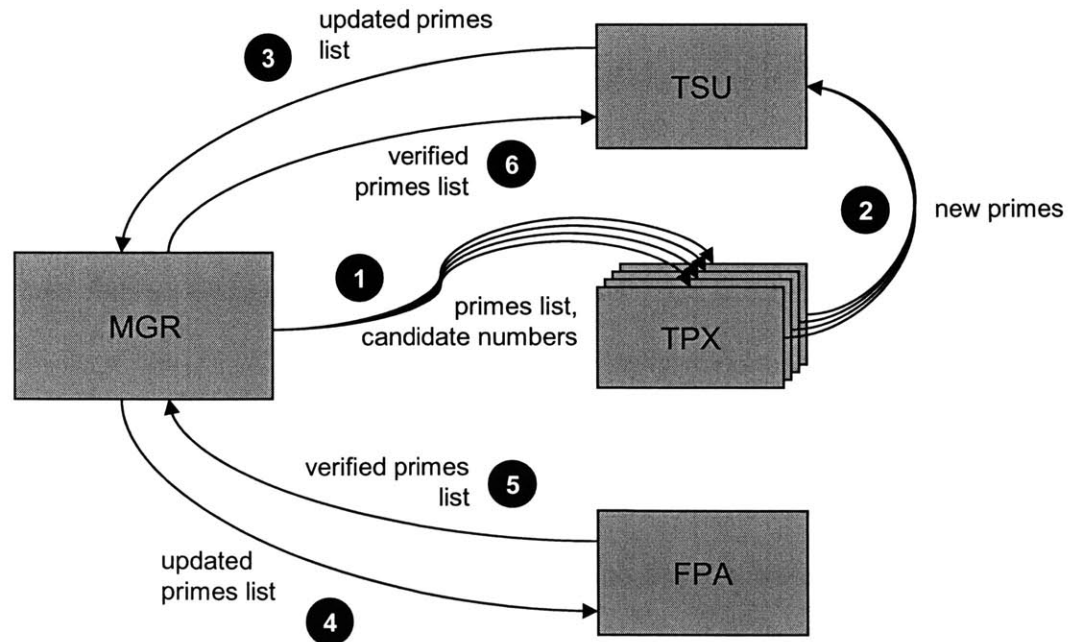


Figure 7. Application Work Flow.

simplified – the manager thread keeps an integer counter which represents the next candidate number, and increments it after assigning the number to a TPX thread. Instead of transmitting data between the threads via an intermediary, as is done in the actual PFS_E design, the TPX threads communicate with the TSU thread directly.

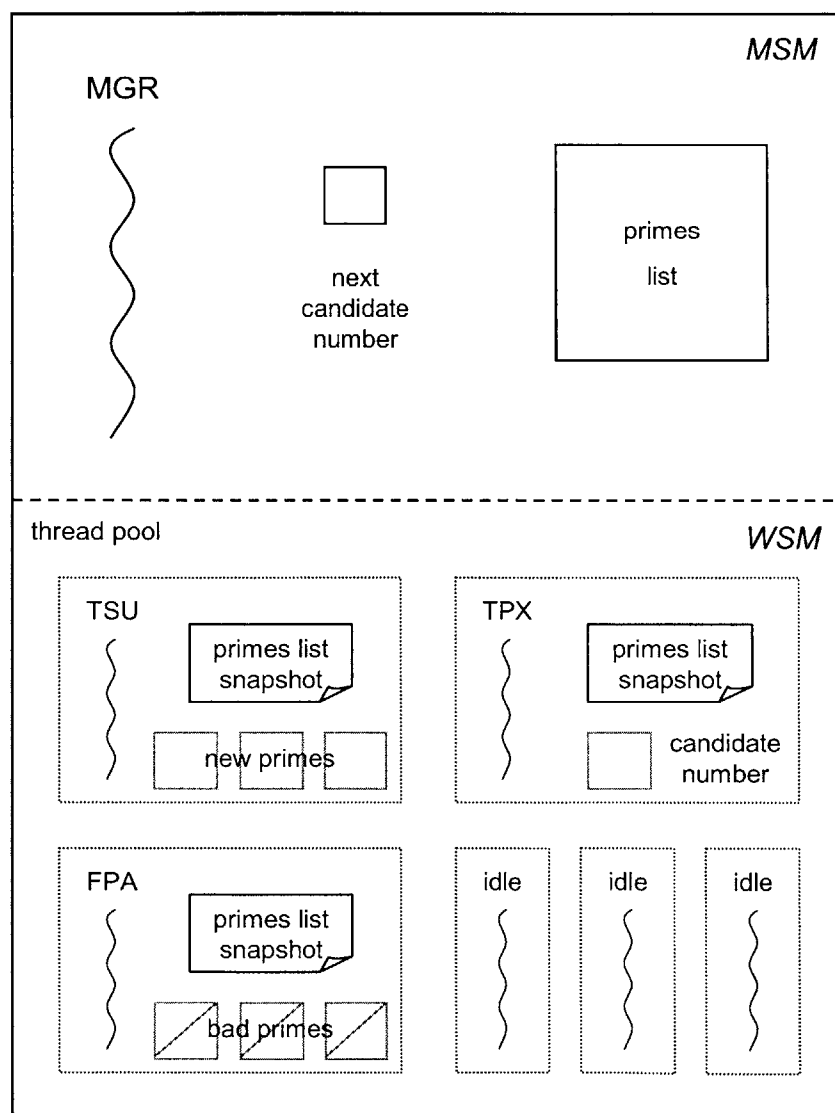


Figure 8. Application Module Thread Structure and Supporting Data Objects.

4.2 Failure Detection Module

A ping-based, non-distributed, adaptive failure detector (FDM) monitors the reimplementation of the PFS_E module. It functions similarly to the detector described in Section 2.3.2: every Δ_i seconds, the detector pings each monitored thread and waits for their replies. If a thread's reply doesn't arrive by the expected time (see Section 4.2.1), the failure detector suspects the thread has failed and notifies the recover/repair module (see Section 4.3). Late arrivals, if occurring before the recover/repair module can act, cause the detector to resume trusting the thread.

4.2.1 Freshness Points

Our failure detector employs *freshness points* to identify late replies, as defined by Chen et al [3]. The k^{th} ping to process p has a freshness point denoted by τ_k , and represents the latest expected arrival time of the reply. Freshness points replace time-outs and essentially serve as expiration dates – if a reply doesn't arrive before its freshness point, the corresponding thread is suspected to have failed. Figure 9 depicts the process graphically.

The freshness point τ_{k+1} for the $(k+1)^{th}$ reply is calculated upon receipt of the previous reply using the *estimated arrival time* EA_{k+1} and *safety margin* α_{k+1} :

$$\tau_{k+1} = EA_{k+1} + \alpha_{k+1} \quad (1)$$

The last n arrival times, where the k^{th} arrival time is represented by A_k , are used to estimate when a ping's reply will arrive. Assuming message transmission delays remain constant, the $(k+1)^{th}$ reply should arrive at EA_{k+1} , its *estimated arrival time* [3]:

$$EA_{k+1} \approx \frac{1}{n} \left(\sum_{l=k-n}^k A_l - \Delta_i \cdot l \right) + (k+1) \cdot \Delta_i \quad (2)$$

The second term, $(k+1) \cdot \Delta_i$, is the next ping's send time. To this, we add the average observed delay for the last n pings, represented by the first term – $\Delta_i \cdot l$ is the send time for the l^{th} ping, and A_l is the corresponding reply's arrival time, making the summation term the time elapsed between the when the ping was sent and when its reply arrived. Changing the value of n affects the estimation's memory – a larger value will use more, older arrival times in the estimation, while a smaller value restricts it to only using recent

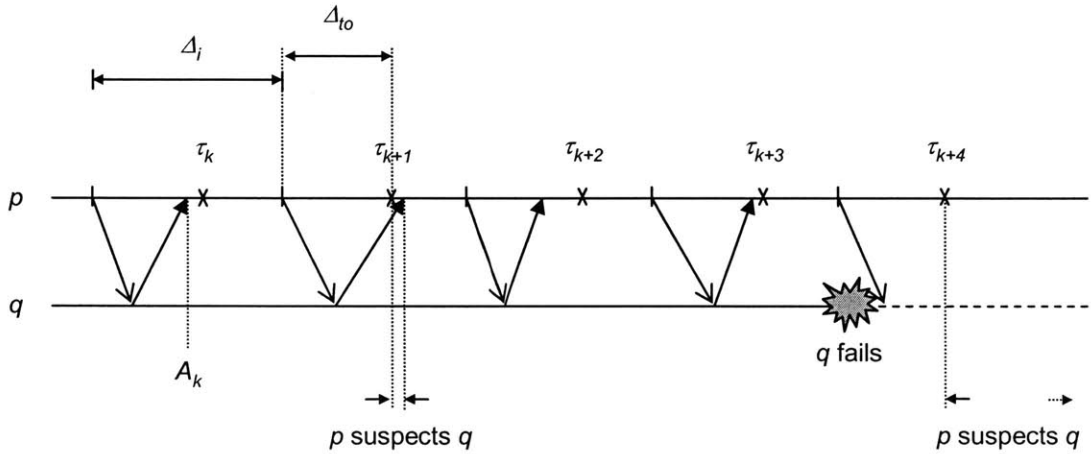


Figure 9. Failure Detection Using Pings and Freshness Points. Process p is monitoring process q , and sends a ping every Δ_i seconds. p waits until the corresponding freshness time point passes before suspecting q .

values. In this way, one can control how dynamic the estimation is and how rapidly it responds to changes in message delays.

Variable transmission delays are accounted for by the safety margin, which is recalculated each time according to a modified version of Jacobson's algorithm [4]:

$$err_k = m_k - a_k \quad (3)$$

$$a_{k+1} = a_k + \gamma \cdot err_k \quad (4)$$

$$v_{k+1} = v_k + \gamma \cdot (|err_k| - v_k) \quad (5)$$

$$rto_{k+1} = \beta \cdot a_{k+1} + \phi \cdot v_{k+1} \quad (6)$$

This algorithm was originally designed to determine a suitable retransmit timer for data transmission across networks. It measures the previous round-trip time, m_k , and uses it to calculate the error, err_k , relative to average round-trip time, a_k (Equation 3). In turn, running estimates of the average and mean deviation, v_{k+1} , are updated using the error, with γ determining its importance (Equations 4 and 5 respectively). A value for the next retransmit timer, rto_{k+1} , is computed from the new average and mean deviation; β and ϕ control how much each quantity influences the final estimate (Equation 6).

For our detector, we modify the algorithm accordingly:

$$m_k = A_k - EA_k \quad (7)$$

$$a_{k+1} = rto_{k+1} \quad (8)$$

Equation 7 calculates the value that would have made the freshness point equal to the actual arrival. This value, or *measured margin*, would have been the optimal safety margin for the previous message and accordingly is an ideal basis for the next one. As noted by Jacobson, taking the absolute value of err_k in Equation 5 causes rapid increases

and gradual decreases in the mean deviation. This behavior is highly desirable for our detector – it anticipates increased delays quickly and avoids converging too fast, both of which can cause false positives.

Bertier et al propose this combination of estimated arrival time calculation and Jacobson’s algorithm in [5]. They prove the resulting failure detector satisfies strong completeness and eventual strong accuracy (see Section 2.1), and falls into the *Eventually Perfect* ($\diamond P$) class of detectors as defined by Chandra and Toueg in [2].

4.2.2 Justification

Since the target application is multithreaded, as opposed to having multiple processes, a distributed detector is unnecessary. Furthermore, a distributed detector would have as many local failure detectors as threads, and each thread would need to run failure detector code in addition to its own. More processor time would be used to run failure detector code, slowing down the application’s progress. As many or more messages would also be transmitted within the process: one thread monitoring $n-1$ other threads requires $2*(n-1)$ messages every period for a non-distributed ping-based implementation; a distributed implementation has each thread monitoring 1 to $n-1$ threads, yielding 2 to $2*(n-1)$ messages per thread and $2n$ to $2n*(n-1)$ messages total every period. These additional messages would consume more bandwidth for message transmission, as well as processor time to process all of them. Our system needs to be as transparent as possible to allow for the actual PFS_E module to remain highly responsive, so these sources of additional overhead are undesirable.

The failure detector uses pings rather than heartbeats also to avoid additional overhead. As seen in Section 2.3.3, a heartbeat-style non-distributed detector requires as many timers as monitored threads, while a ping-based detector needs only one. Even though ping-based detectors tend to be less accurate, they incur less overhead and are less complex, making this an acceptable trade-off.

4.3 Recover/Repair Module

Any thread that the failure detection module suspects as failed is queued for repair in the recover/repair module (RPM). Repair requests in the form of suspected threads are added to its queue as they are received. Whenever there are requests in the queue, the first one is removed and processed. During repair, the module ensures the thread in question is still suspected – if this isn't the case, the thread is de-queued. A new thread is created to take the suspected one's place, and receives the work that was in progress at the time of failure. Since the FPA thread removes duplicates from the primes list, we don't need to guarantee a work request is processed exactly once.

Chapter 5: Implementation

In this chapter, we expand on the design by describing in greater detail the system as a whole and the three components individually. We first make a distinction between the abstract notion of modules and the implementation construct of threads. The remainder of the chapter deals with the inner workings of each of the threads, paying particular attention to resource management, communications, and algorithms.

5.1 Modules vs. Threads

Up to this point, the system has been described largely in terms of modules, where each module deals with performing a specific task and consists of the associated code and data structures. Their corresponding threads, however, do not execute entirely within these confines. A portion of the code in each module provides a means for another module to interface with it, as depicted in Figure 10. This *interface code*, or *icode*, is used by that interfacing module's associated thread(s) and allows restricted access to functions and data in the parent module. In our system, the MSM has *icode* granting worker threads access to their dataglobs, the FDM's *icode* allows worker threads to support failure detection, the RPM provides the failure detection thread (FDT) with *icode* to submit repair requests, and the MSM contains *icode* which performs repairs at the request of the recover/repair thread (RPT).

This scheme creates a clear delineation in the modules and serves as an abstraction barrier: the modules know only how to interface with each other, and the details of their implementations are hidden and don't affect these interfaces.

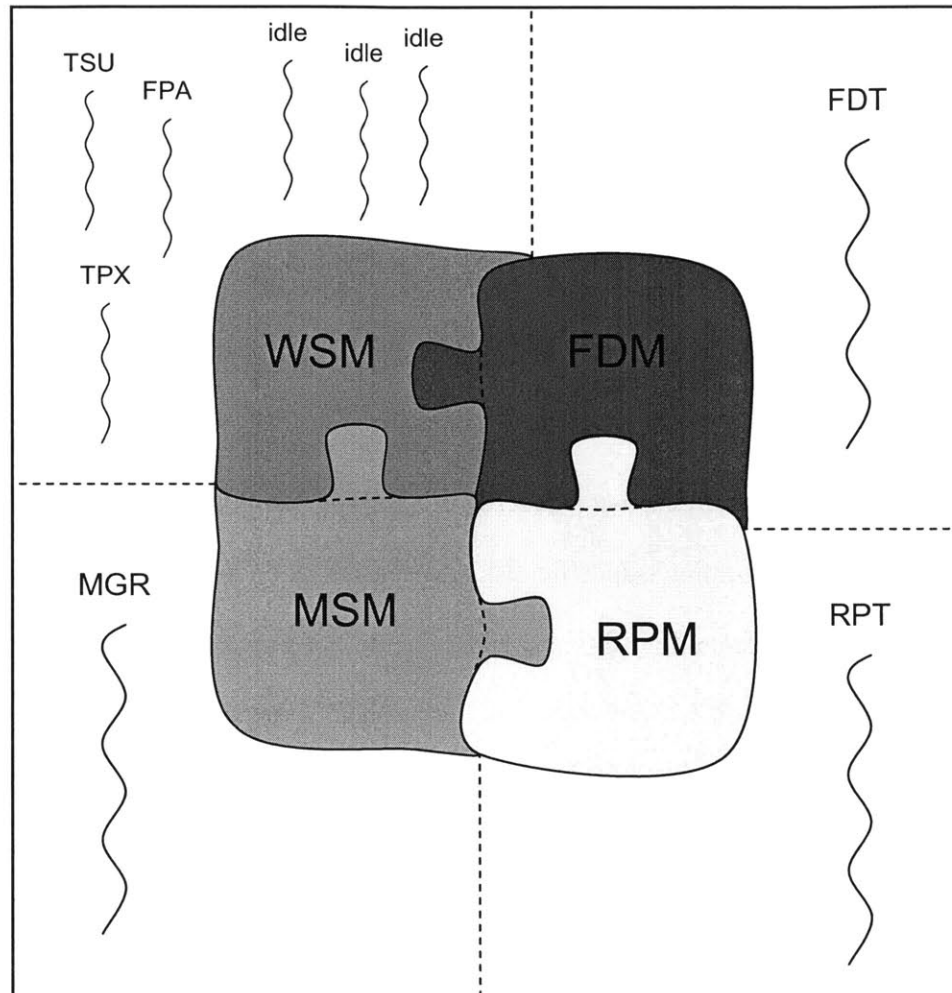


Figure 10. Breakdown of System Modules and Threads. Each colored piece represents one module in the system, as defined by their three-letter acronyms. The nubs connecting the modules together represent the icode. Accordingly, threads execute within the confines of the dashed boundaries and are also identified by their three-letter acronyms.

5.2 Application Threads

During system initialization, the manager thread performs two main tasks – it prepares the data structures needed during execution and it creates all the necessary threads. If enabled, the failure detection and recover/repair threads are created as well and allowed to initialize before the application’s actual work begins. The manager thread’s remaining tasks involve efficiently managing these resources.

5.2.1 Data Management

A simple array with its length stored at the first index serves as the primes list. This structure allows efficient cloning, since we can dynamically allocate additional memory without having to re-compute the list’s length. Whenever the manager thread receives a updated primes list, an new snapshot is given to each worker.

Aside from work-related data, the manager thread maintains two variables that serve as status flags for the failure detection and recover/repair threads. The manager thread can notify the threads when to terminate via these variables.

Arrays of all thread IDs, the *thread list*, and of pointers to each worker’s data structure, the *worker’s data list*, are also maintained. The thread list provides a mapping of array index to thread ID that proves useful when performing thread repairs (see Section 5.4). Furthermore, referencing threads only by this index rather than their thread ID eliminates the need for a reverse mapping – if IDs were used, then a mapping of thread ID to thread index would be needed to access data in the worker’s data list.

The worker's data list contains pointers to data structures called *dataglobs*, which are prepared and managed by the manager thread (see Figure 11). Each worker thread has one of these structures, and accordingly the list is indexed by thread index. Dataglobs contain two types of data – thread-specific data, and work-related data. The manager thread's thread ID, the associated worker thread's thread index, its status as specified by the manager, and its thread type comprise the thread-specific data. A thread's type defines what kind of worker thread it is – TPX, TSU, or FPA – while its status functions similarly to the failure detection and recover/repair threads' status variables. Workers, however, can be set to idle in addition to work or exit.

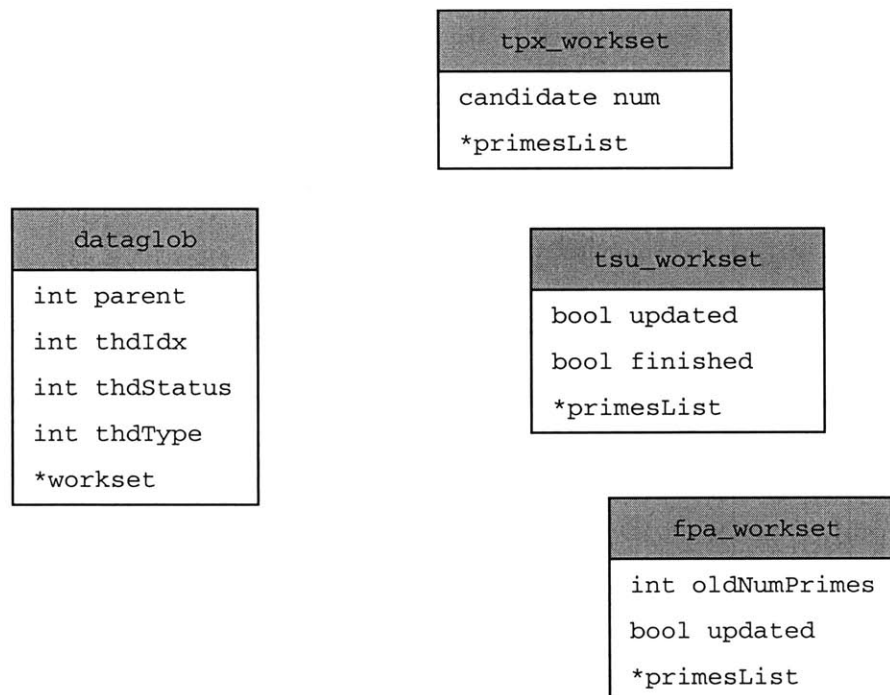


Figure 11. Data Structures Used in the Application. Each worker thread has a dataglob object, maintained by the manager thread. In addition to thread-specific information, a dataglob contains work-related data located in a workset. Each type of worker has its own specific type of workset.

The work-related data found in dataglobs, called a *workset*, is specific to the type of worker. Each workset contains a pointer to a snapshot of the primes list. A TPX thread's workset also contains its candidate number. Boolean variables in the TSU thread's workset, `updated` and `finished`, allow notification of updated data or a lack of new data, respectively. Both Booleans are necessary because the manager thread clears `updated` once it has received the new data from the TSU thread; `finished` is only changed by the TSU thread, allowing its value to persist until the next update. The FPA thread also uses an `updated` variable, as well as an integer that stores the length of the primes list the last time it ran, `oldNumPrimes`.

5.2.2 Thread Management

At initialization, the manager thread creates the pool of workers and passes each of them a pointer to their respective dataglob. Thread management occurs via this shared memory, with the manager thread instructing workers when to idle, when to work, and when to exit. In the manager thread's main loop, its management duties are broken down into three subtasks, one for each worker type:

```

while nextNum < MAXNUM or !TSU/finished,
    check FPA for updates and act accordingly
    check TSU for updates and act accordingly
    if TPX(currIdx) is idle, assign it work
    yield

```

We remain in the main loop while there are still candidate numbers to be checked for primality, or the TSU thread received new data on its last update. The second condition

ensures all assigned candidate numbers have a chance to get checked and be sent to the TSU thread before the program exits.

During each iteration of the main loop, the manager thread looks at a different TPX thread using *currIdx*:

```
if TPX(currIdx) is idle,  
    update its primes list snapshot  
    assign it nextNum  
    increment nextNum  
    set it to work  
  
increment currIdx
```

If that thread is idle, we update its workset with a new snapshot and its assignment, and set it to work.

After the TSU thread performs an update, the manager thread must acquire the new data:

```
// check TSU for updates  
if TSU/updated,  
    // get updates from TSU  
    update our copy of primes list  
    clear TSU/updated  
  
    // run FPA to verify updates  
    set TSU to idle  
    update FPA's copy of primes list  
    set FPA to work
```

On its next update, the TSU thread receives the new data and updates its primes list. In turn, the manager updates its local copy of the primes list and clears the updated flag. Next, verification of the new primes list occurs by giving the FPA thread a snapshot and setting it to work. The TSU thread needs to be idled to give verification a chance to finish before another update occurs.

When the FPA thread returns to idle status, it has completed verification and may have updated data for the manager:

```
//check FPA for updates  
if FPA is idle,  
    if FPA/updated,  
        // get new data  
        update our copy of primes list  
        update TSU's copy of primes list  
        clear FPA/updated  
  
    //resume TSU  
    if TSU is idle,  
        set TSU to work
```

If the FPA thread has indeed updated the primes list, the manager thread updates its local copy as well as the TSU thread's copy, and clears the updated flag to notify the changes have been received. Whether the primes list was updated or not, the TSU thread must be set back to work, causing updates to resume.

5.2.3 Manager/Worker Interface

The manager sub-module provides worker threads with limited access to their dataglobs using supplied *icode*. Fashioned as a wrapper for all worker threads, this *icode* coordinates with the manager thread to achieve common management of all worker threads, and serves as their entrypoint.

At initialization, worker threads prepare internal communications between TPX and TSU threads. The main loop common to all workers is then entered:

```
wait while status is set to idle  
  
if status is set to work,  
    dispatch to common work function  
    set status to idle  
else, if status is set to exit,  
    return
```

Thread management on the worker's side occurs here – the thread waits until it isn't set to idle, then either performs its work and returns to idle status, or exits. Transitioning back to idle status allows the manager thread to detect when work has completed. The common work function, in turn, dispatches the necessary data to the specific work function:

```
if thread type is TPX,  
    dispatch workset and socket to TPX work function  
    wait a random number of seconds  
else, if thread type is TSU,  
    while status is set to work,  
        dispatch workset and socket to TSU work function  
        wait 12 seconds  
else, if thread type is FPA,  
    dispatch workset to FPA work function
```

All thread types have their corresponding worksets passed to their work functions. TPX threads and the TSU thread also receive opposite ends of a socket through which discovered primes are sent. The FPA thread immediately returns to the worker main loop once completing its work. However, TPX threads have a forced delay – this is intended to simulate the longer and varying computation time of actual TPX threads. Similarly, a 12 second delay is imposed on the TSU thread, thereby forcing updates to occur

approximately every 12 seconds. Another loop encapsulates the TSU thread and this delay, causing continual updates while its status is set to work.

5.2.4 Prime Discovery

Primality is determined by checking if a number is prime relative to the list of known primes, as shown in the pseudocode for TPX threads:

```
isPrime = true  
  
for each element primei in the primes list,  
    isPrime &= !(number % primei == 0)  
  
if isPrime,  
    send number to TSU via shared socket
```

Since elements in the primes list are kept in increasing order, this method essentially finds a number's prime factorization and rejects those numbers with factors in addition to 1 and itself. Any discovered prime is sent to the TSU thread via a socket shared amongst all TPX threads.

About every 12 seconds, the TSU thread checks the socket for new primes:

```
while the socket isn't empty,  
    copy primes from the socket into a buffer  
    update the number of new primes received  
  
if no new primes received,  
    set TSU/finished  
else,  
    clear TSU/finished  
    increase the size of the primes list to accommodate the new primes  
    copy the new primes into the primes list  
    update the length counter for the primes list  
    set TSU/updated
```

If no new primes are received, the finished flag is set and the thread waits for the next update. Otherwise, additional memory is allocated to the primes list for holding the new primes and the length counter is updated. We make sure to clear the finished flag and set the updated flag before returning.

The following pseudocode depicts how the FPA thread verifies new data:

```

sort the primes list
set the sorted flag if the list was reordered

for each element  $elt_i$  in the primes list,
    check if  $elt_i$  is prime relative to primes  $elt_1$  to  $elt_{i-1}$ , inclusive

    if  $elt_i$  isn't prime,
        set edited flag
        increment numBadPrimes
    else if edited,
        slide  $elt_i$  over by numBadPrimes

if edited,
    shrink primes list by numBadPrimes elements
    update the length counter in the first element of primes list
    set FPA/updated
else if sorted,
    set FPA/updated

save primes list length counter in FPA/oldNumPrimes

```

First, the primes list is sorted using an insertion sort. Since we keep the number of primes from the last execution, we only need to find where the new primes belong in the list. Then the list is verified using the same principle that TPX threads' discovery algorithm uses: we ensure each element is not divisible by any element that comes before it in the list, thereby ensuring its prime factorization consists of 1 and itself and that it is indeed prime. By only having to compare each element to those before it, we are able to reduce the number of comparisons needed to verify the entire list. Any errant primes are

removed by shifting over all subsequent elements and later readjusting the size of the list. If bad primes are removed, or the list is reordered, we set the updated flag and return.

The primes list needs to be sorted before verification because there is no guarantee on the order of discovery. For instance, assume three TPX threads are checking the numbers 2, 3, and 4. If 4 gets checked first, it will incorrectly be seen as prime – the primes list is empty at this point. The ordering of the primes list would be {4, 2, 3} once these threads finish, and all three numbers would pass verification since the routine relies on a sorted list. Sorting could be eliminated if the routine was modified to check each element against every other element, but this wouldn't yield any gains in running time. Defining n to be the current length of the list and old to be the prior length, $oldNumPrimes$, insertion sort takes $O(n*(n-old))$ time, plus $O(n*(n+1)/2)$ time for verification. The altered verification routine would require $O(n^2)$ time. While algorithmically equivalent, the altered routine won't run faster in practice, and will likely run slower than our routine because of the constant factors.

5.3 Failure Detection Thread

Once created by the manager thread, the failure detection thread initializes its estimation data, prepares its communications, and begins its timer. A warm-up period, lasting long enough for $2n$ pings to be sent, expires before the manager thread starts delegating work and failure detection begins; this delay allows the failure detector time to stabilize its estimations. We examine data management issues, communications mechanisms, and timing details of the process in this section.

5.3.1 Data Management

In addition to the necessary estimation data, the failure detector maintains an array of each monitored thread's state, the *thread state list*, which holds information regarding thread activity. There are four possible states – *alive*, *pending*, *suspected*, or *killed* (see Figure 12). *Alive* signifies that the last ping sent to the thread was replied to; all threads start out in this state. Once pings are sent out, all threads transition to the *pending* state, meaning a reply is pending. The receipt of a reply causes the thread to return to the *alive* state. However, if the corresponding freshness point expires, the detector suspects the thread to have failed and marks it as *suspected*. A message is sent to the recover/repair thread at this point, and the suspected thread is queued for repair (see Section 5.4). If a thread's reply arrives while it is marked as *suspected*, it transitions to *alive* and will be

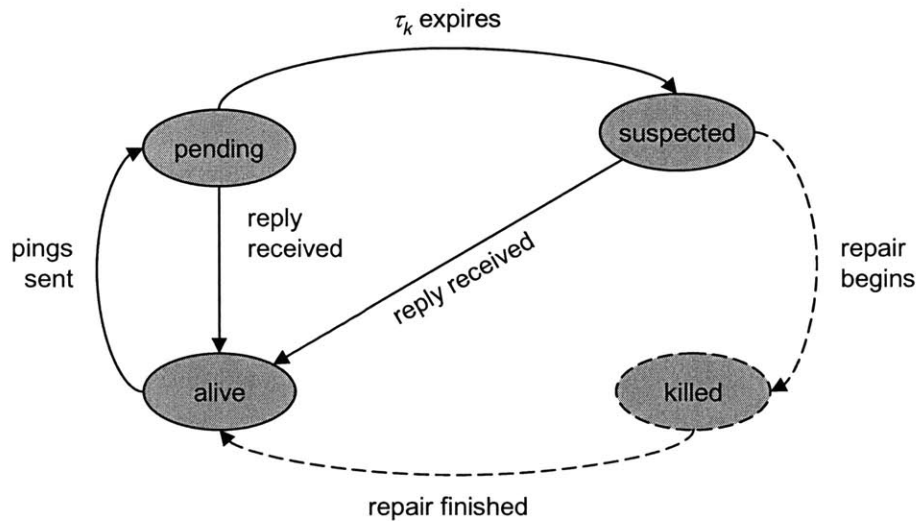


Figure 12. Finite State Machine Representing Possible Thread States and Transitions. Regions with solid lines represent actions and states associated with the FDT, and those with dashed lines deal with the RPT.

de-queued in the recover/repair thread. Otherwise, the recover/repair thread sets the thread to *killed*, carries out repairs, and sets the thread back to *alive*. The states *suspected* and *killed* are both necessary because a reply may arrive late, and we need a way to distinguish between when a thread has merely been suspected and when repairs have begun.

5.3.2 Communications

All failure detection-related communications in the system is accomplished using POSIX real-time signals. Also called software interrupts, these short, quickly-delivered messages are managed by the kernel. They can be sent from one thread (or process) to another, interrupting the destination thread and invoking a signal handler. Furthermore, values can accompany signals, allowing the recipient to distinguish the source.

In our system, arranging to handle the appropriate signals and block others allows any thread to be monitored. All threads perform this registration using the failure detection module's *icode*. However, threads wishing to be monitored must prepare to receive pings using the provided signal handler, while all other threads need to block these signals to keep from interfering.

There are three main signals used in failure detection – `PING_SEND`, `PING_REPL`, and `THD_REPAIR`. `PING_SEND` is sent from the failure detection thread to all monitored threads when the interval timer expires. On receipt of a `PING_SEND` signal, the receiving thread invokes its signal handler to process it, sending back a reply with its thread index via `PING_REPL`. The failure detector processes the replies and computes each thread's subsequent freshness point using the procedure in Section 4.2.1. Any expired freshness

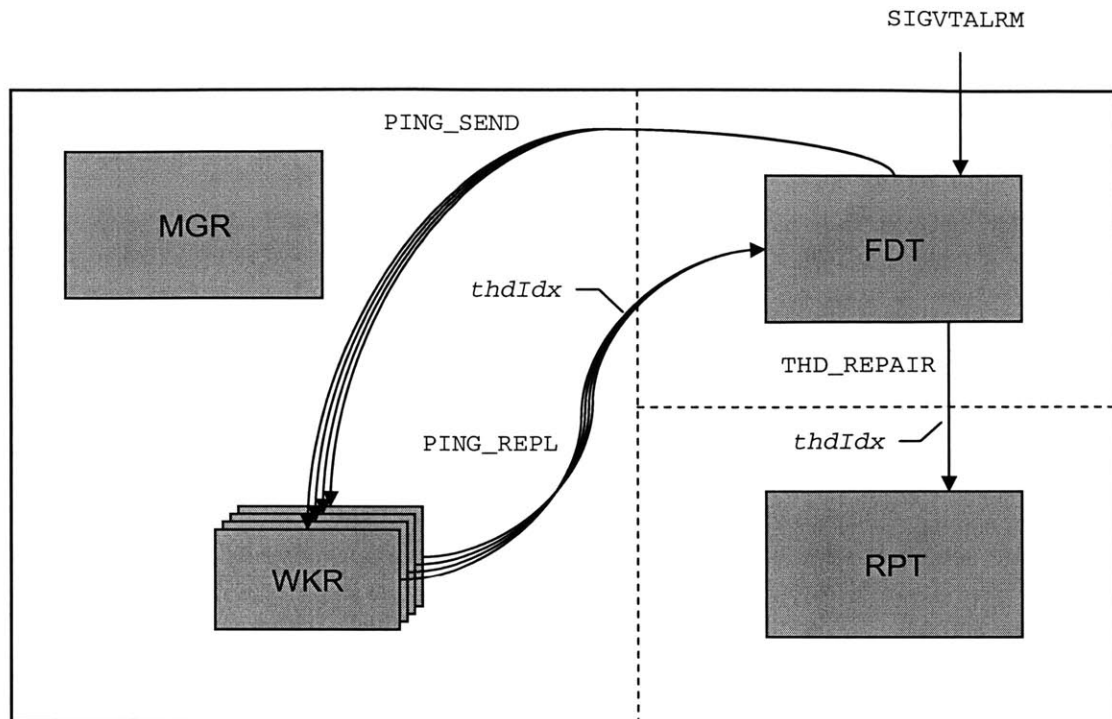


Figure 13. Communication Among Threads via Signals. `SIGVTALRM` signifies a timer expiration event, described in Section 5.2.3.

point causes the failure detection thread to execute the recover/repair module's *icode*, which requests repair on the corresponding thread by sending a `THD_REPAIR` signal to the recover/repair thread. Figure 13 depicts how the threads communicate via these signals.

5.3.3 Timing & Monitoring

The failure detection uses thread CPU time in all timing-related tasks, including when to suspect a thread. Operating system-provided interval timers, also using thread CPU time,

notify the detector when to send pings by dispatching a virtual-time alarm signal, or SIGVTALRM. Real-time is undesirable for failure detection, since we only care about thread activity relative to the process, not to the entire computer; a late reply using real-time could easily be the result of some other process hogging the processor. Process CPU time could work, but we'd still have the same problem on a smaller scale – a thread could hog the processor³.

Estimated arrival times are calculated using an optimized version of Equation 2:

$$old = A_{k-n} - (k - n) \cdot \Delta_i \quad (9)$$

$$new = A_k - k \cdot \Delta_i \quad (10)$$

$$sum_{k+1} = sum_k - old + new \quad (11)$$

$$EA_{k+1} = \frac{1}{n} \cdot sum_{k+1} + (k + 1) \cdot \Delta_i \quad (12)$$

Instead of summing the arrival times each time we calculate a freshness point, we save the sum and simply update it by removing the old value and adding the new one. The gains realized by this modification should be significant for large values of n .

In practice, the send times aren't guaranteed to be exactly on integer multiples of the polling interval. There are also non-zero delays in processing the signals. Since the estimated arrival time is calculated using the theoretical send time and the actual arrival time, the average observed delay term accounts for all of these delays – the ping's send delay, the reply's send delay caused by processing delays, and transmission delays (see Figure 14).

³ The detector's *getTime()* function is implemented using the *getrusage()* system call. Most versions of Linux have incomplete implementations of this call – when querying for the process CPU time in a multithreaded program, only that thread's CPU time is given. Furthermore, the interval timers provided by the operating system seem to use the same way of determining process time, since they expire in sync with thread CPU time. Working around both of these issues would be complicated and likely wouldn't improve results.

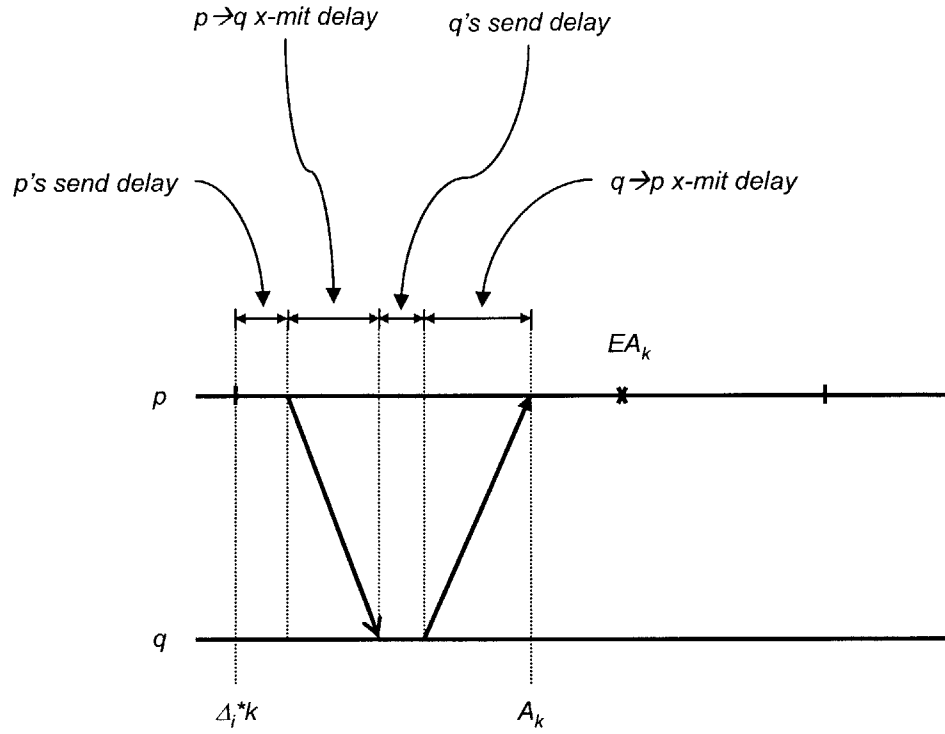


Figure 14. Observed Communication Delays. Process p is monitoring process q . Here, we look at one particular ping and reply, paying attention to send and transmit (x-mit) delays experienced in the actual implementation.

Detection of late replies occurs in the main loop of the failure detection thread:

```

while status is set to work,
  wait until pings are sent out
  do until no more threads' replies are pending,
    for each monitored thread  $thd_i$ ,
      if current time >  $thd_i$ 's freshness point,
        set  $thd_i$ 's state to suspected
        notify RPM of suspected failure

```

Control remains in the main loop until the manager thread of the application changes the detector's status. The thread idles until pings are sent out, at which point it begins checking for lateness. Until all monitored threads either are suspected or deliver timely

replies, we check pending threads' freshness points for expiration. Those that have expired result in the detector setting their state to *suspected*. Additionally, the failure detection thread sends a repair request to the recover/repair thread.

5.4 Recover/Repair Thread

Initialization in the recover/repair thread primarily consists of waiting until the failure detector has warmed up. As is done for the failure detector, all threads must register themselves with the recover/repair thread via the provided *icode* to block the `THD_REPAIR` signal, thereby avoiding conflicts.

The primary data structure in the recover/repair module is the *repair queue*, which holds suspected threads awaiting repair, referenced by thread index. An array with a head and a tail pointer implements the queue, and threads are en-queued in the signal handler for `THD_REPAIR`. They are subsequently processed in the thread's main loop:

```

while status is set to work,
    wait while queue is empty
    if thread at the head of the queue,  $thd_{head}$ , is set to suspected,
        set  $thd_{head}$ 's state to killed
        repair  $thd_{head}$ 
        update FDM data for  $thd_{head}$ 
        set  $thd_{head}$ 's state to alive

```

As with the failure detector, the recover/repair thread remains in its main loop until it is instructed otherwise. If the queue is non-empty, we make sure the head thread is still suspected before trying to repair it. Repair begins when its state is set to *killed*; this serves as notification to other threads that it is being repaired. An application-supplied procedure handles the actual repair, and upon its completion, any failure detector data

associated with the thread needs to be updated. Namely, the next freshness point needs to be calculated, since the previous reply never arrived. Once the thread's state is set to *alive*, repair is complete and the thread can resume work.

The manager sub-module of the application provides the repair procedure, which is also part of the *icode* interfacing the two modules. It simply creates a new thread to replace the failed one, passing it the same dataglob. Since we access threads by their index in the thread list, we avoid having to update any references to the failed thread by reusing this index for the replacement thread.

Chapter 6: Performance

Our system has been implemented in multithreaded C++ using POSIX threads. Development and performance testing was done on a 400 MHz Intel Pentium II with 128MB RAM, running Red Hat Linux release 8.0. In this chapter, we examine how effectively it meets our goals. Specifically, the speed and accuracy of the failure detector is evaluated, as is the recover/repair module's effectiveness in performing repairs. We conclude with an analysis of the overhead incurred for failure detection and repair.

6.1 Failure Detection Module

Failures need to be caught quickly by the detector, and false positives need to be avoided; any mistakes will likely be repaired by the recover/repair module before they can be corrected, slowing down the application's progress by terminating useful work and wasting time on unnecessary repairs. As such, our detector needs to perform well in terms of detection time and average mistake rate – the other metrics aren't as important. Minimizing these two values will serve to keep the system responsive and efficient.

For tests in this section, the application module is configured to have 10 worker threads and the recover/repair module is disabled – we are only interested in the failure detector's performance. Suggested values for the detector's parameters [5] are the following: $\Delta_i = 5$ seconds, $n = 1000$, $\gamma = 0.1$, $\beta = 1.0$, $\phi = 2.0$. However, these values are more or less defaults and won't necessarily provide the best performance for our system.

We also know how the detector will be used and the kind of performance we need. Accordingly, the following subsections examine the effect of each of the detector's parameters on its performance, and determine appropriate values to provide our desired quality of service.

6.1.1 Estimated Arrival Times

Revisiting Equation 2, there are two parameters affecting the estimated arrival time – the polling interval, Δ_i , and the number of messages to include in the calculation, n . The polling interval plays a large part in determining the detection time. A significantly smaller value for Δ_i than suggested is needed, since we want failures to be detected quickly and, more importantly, our detector uses thread CPU time rather than real-time as its local clock. Preliminary tests show the observed communication delay, or the delay between when pings are sent and when replies are received, can reach 0.1 seconds and averages around 0.04 seconds. Making the interval less than 0.1 seconds could allow a new set of pings to be sent out before the previous one's replies are received. Additionally, using an interval of 1 second or greater translates into a set of pings being sent every several 12-second updates – this is much too infrequent to provide efficient failure recovery. These data suggest a conservative minimum Δ_i of 0.25 seconds and an absolute maximum of 1 second. We settle on $\Delta_i = 0.5$ seconds to limit the message bandwidth used.

Increasing the value of n causes the estimated arrival time to have a smoother graph over time, as shown in Figure 15. To simplify the graph, we ignore the $(k+1)*\Delta_i$ term, making the y-coordinates represent the average observed delay – this is the only portion of the estimated arrival time calculation that is affected. If we use the suggested value of $n = 1000$, not only would it take a long time for the detector to warm-up, but the observed delay would be virtually constant barring drastic changes in system behavior. Even with $n = 50$, there is very little variation in the delay. To keep the estimation

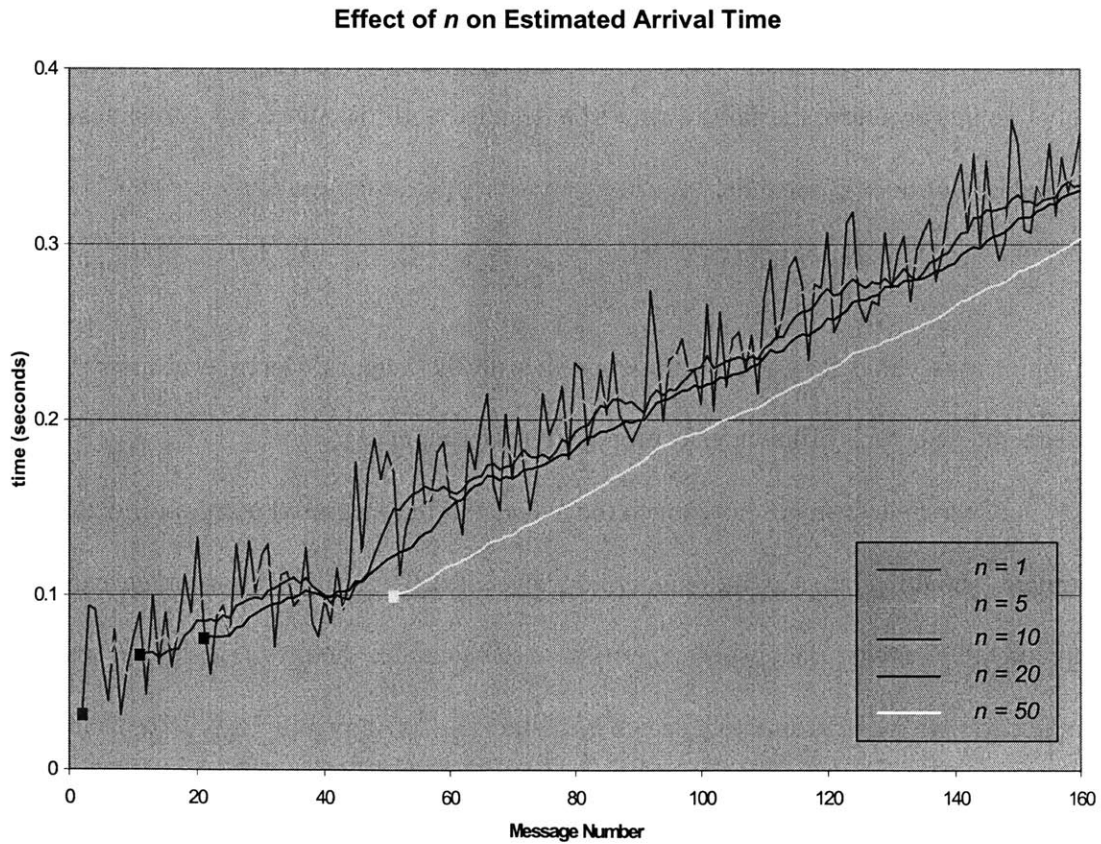


Figure 15. Graph of n 's Effect on Estimated Arrival Time.

dynamic, we choose $n = 10$ – any recent trends in the observed communication delay are reflected in the resulting estimate without being too erratic or too smooth over time.

6.1.2 Safety Margins

While the estimated arrival time serves as a rough basis for a freshness point, the safety margin allows us to fine-tune the value. By tweaking the three constants involved in its calculation – β , ϕ , and γ – we can configure the detector to provide fast detection times and minimal false positives. We look at the worst-case detection time since there are no actual failures, where the worst-case is when a thread crashes immediately after sending a reply. In this scenario, the failure won't be detected until the subsequent freshness point – a duration of over Δ_i seconds:

$$T_D^{\max} = \tau_{k+1} - A_k \quad (13)$$

Graphing this value for each message along with the actual difference in arrival times reveals the detector's efficiency, as proposed by Bertier et al [5].

β , which determines how much the average of the measured margin influences the estimate, should remain at the suggested value of 1.0 – scaling an average can have unpredictable results. Jacobson's algorithm also calculates the mean deviation, which in conjunction with ϕ appropriately scales the estimate. As shown in Figure 16, changes in ϕ cause translations in the graph of the maximum detection time. We set the parameters as follows: $\Delta_i = 0.5$ seconds, $n = 10$, $\beta = 1.0$, $\gamma = 0.1$. The suggested value of 2.0 succeeds in providing a good detection time at the expense of several false positives. Jacobson suggests a value of 4.0 in [4], which for our system avoids nearly all false

positives but at the cost of an extremely conservative detection time. A value of 3.0 seems aggressive enough in terms of detection time while avoiding most false positives. Table 1 summarizes the performance characteristics of each configuration.

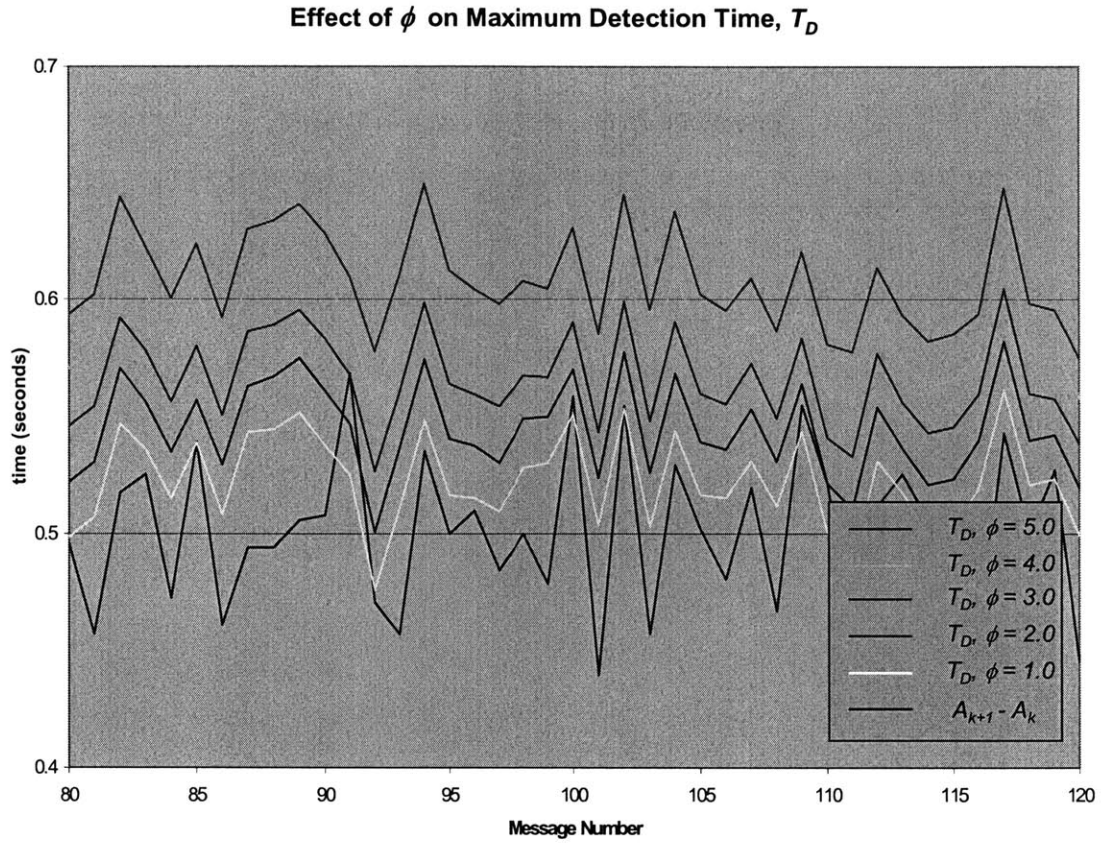


Figure 16. Graph of ϕ 's Effect on Maximum Detection Time. Five monitored TPX threads are given different values for ϕ , and their maximum detection time is graphed as a function of message number. Here we show messages 80-120 to highlight the differences.

Changes in γ , on the other hand, primarily change the amplitude of the graph in Figure 17. We set the parameters as follows: $\Delta_i = 0.5$ seconds, $n = 10$, $\beta = 1.0$, $\phi = 4.0$. This behavior is expected, since γ is the gain for the error in the last measured margin. The suggested value of 0.1 works well, while larger values offer no performance benefits and can even result in false positives. Average detection times and mistake rates for the different configurations are presented in Table 2.

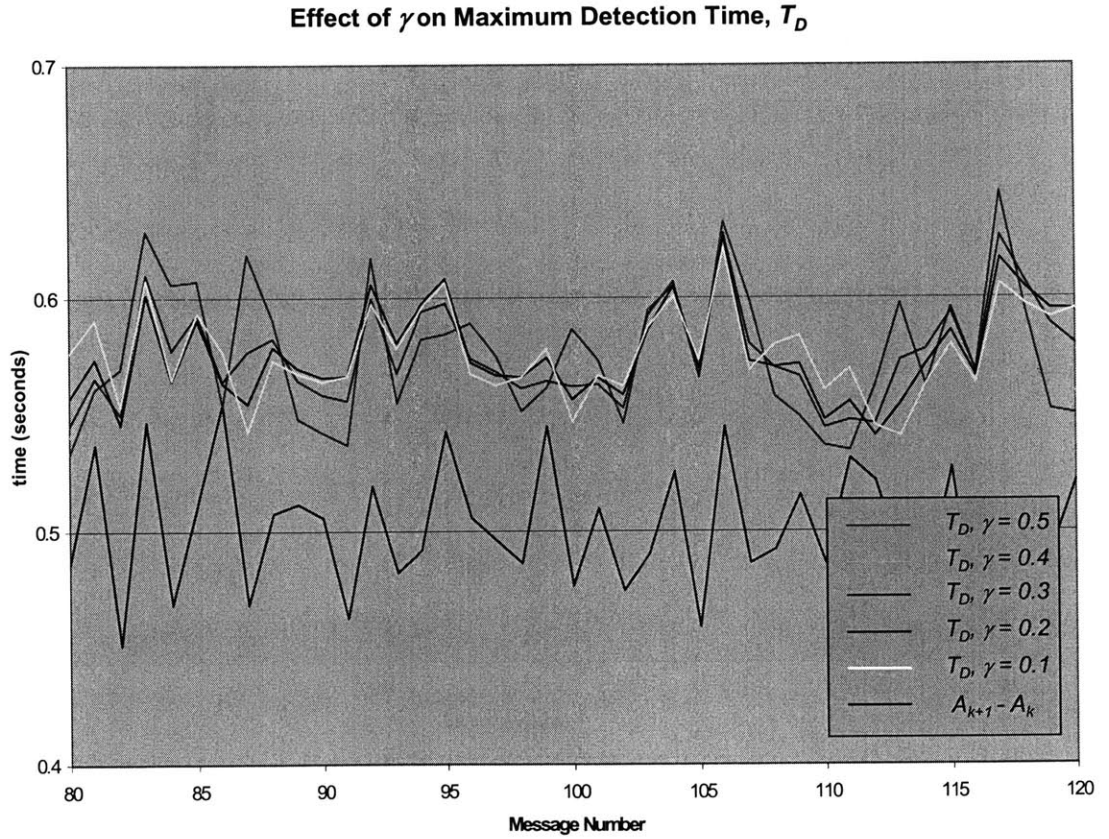


Figure 17. Graph of γ 's Effect on Maximum Detection Time. Five monitored TPX threads are given different values for γ , and their maximum detection time is graphed as a function of message number. Here we show messages 80-120 to highlight the differences.

Table 1. Effect of ϕ on Maximum Detection Time and Average Mistake Rate.

ϕ	T_D (seconds)	λ_M
1.0	0.525	0.200
2.0	0.547	0.031
3.0	0.569	0.013
4.0	0.591	0.006
5.0	0.612	< 0.006

Table 2. Effect of γ on Maximum Detection Time and Average Mistake Rate.

γ	T_D (seconds)	λ_M
0.1	0.579	< 0.006
0.2	0.581	< 0.006
0.3	0.582	< 0.006
0.4	0.583	< 0.006
0.5	0.585	0.006

Based on the data, we set β , ϕ , and γ to 1.0, 3.0, and 0.05 respectively. We run the application and detector for nearly 12 hours, generating primes up to 100000 (see Figure 18). The average maximum detection time is 0.561 seconds with a total of 91 false positives for 2734 messages, yielding an average mistake rate of 0.033 for the system as a whole and 0.003 per thread.

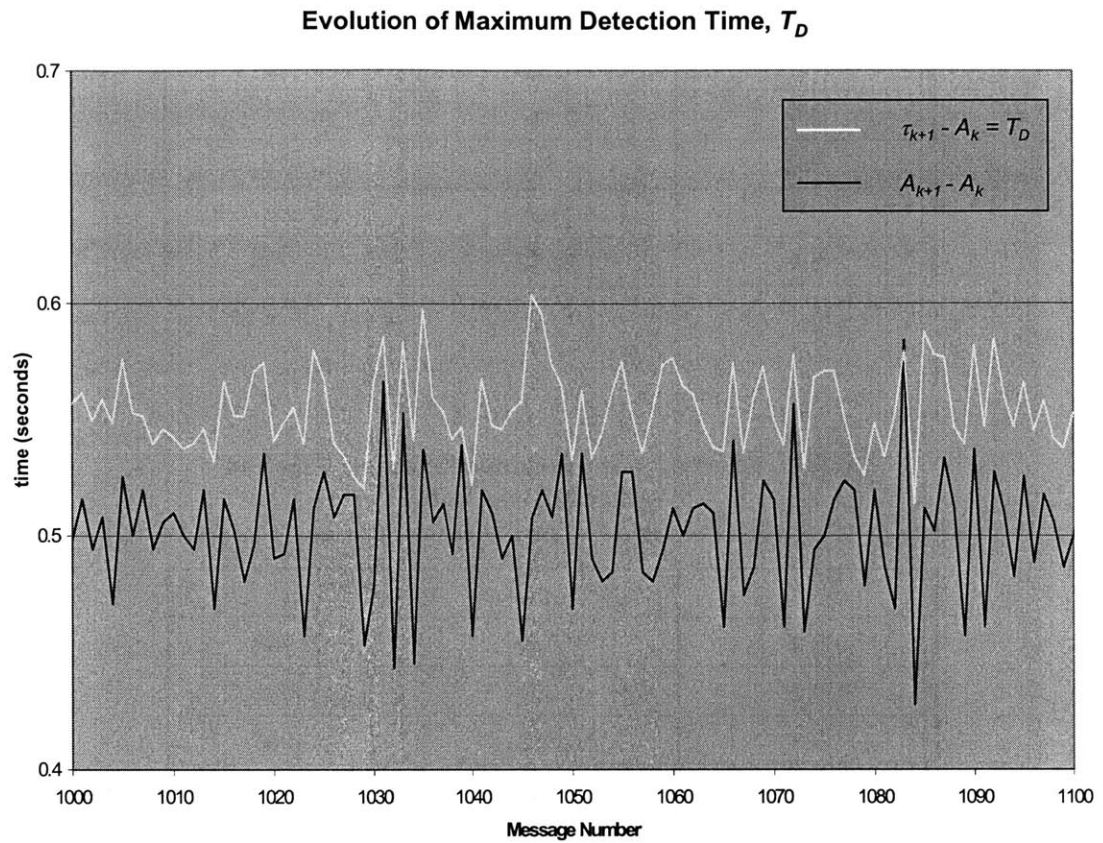


Figure 18. Graph of Maximum Detection Time with Optimized Settings.
 A single TPX thread's maximum detection time is shown for messages 1000 to 1100.

6.2 Recover/Repair Module

Using the detector settings established in Section 6.1, we enable the recover/repair module and evaluate its performance. In particular, we look at actual detection times and how long repairs take. To facilitate this evaluation, worker threads are induced to fail by randomly having them self-terminate with a small probability. The common worker pseudocode (Section 5.2.3) is modified as follows:

```
if thread type is TPX,  
    dispatch workset and socket to TPX work function  
    wait a random number of seconds  
    self-terminate with a small probability  
else, if thread type is TSU,  
    while status is set to work,  
        dispatch workset and socket to TSU work function  
        wait 12 seconds  
        self-terminate with a small probability  
else, if thread type is FPA,  
    dispatch workset to FPA work function  
    self-terminate with a small probability
```

Before terminating, the worker thread sends a THD_OBIT signal to the failure detector; this allows the system to record the time of death in the detector's time frame for performance measurements. Similarly, the recover/repair thread sends the detector a THD_BIRTH signal to record the time of birth.

Running all components with a maximum candidate number of 5000 and a worker self-terminate probability of 99.5%, there are 29 failures and no false positives. The average observed detection time is 0.379 seconds, and the average repair time is 0.035 seconds.

6.3 Overhead

To evaluate how much the failure detector and recover/repair module slow down the application, we examine the CPU times for each thread (see Table 3); failure induction is disabled for these tests. Time spent on worker threads increases 9.4% due to executing failure detector *icode*. A higher polling interval will cause this difference to increase – the more often pings are sent, the more often *icode* is executed. The manager thread sees a similar increase, likely a direct result of the worker threads’ increase – if the worker threads remain active longer, the manager thread must also remain active longer to manage them. With the detector and recover/repair threads, the total increase in CPU time is 28.4%. While this total increase is not minor, the application itself only suffers a 9% overhead from failure detection.

Table 3. Effect of FDM and RPM on CPU Times.

<i>Thread Type</i>	<i>Application CPU Time (seconds)</i>	<i>App + FDM + RPM CPU Time (sec.)</i>	<i>% Increase</i>
MGR	68.580	74.469	8.6%
WKR	677.695	741.207	9.4%
FDM		73.789	
RPM		69.047	
Total	746.275	958.512	28.4%

Chapter 7: Related Work

The vast majority of publications on failure detectors are geared towards monitoring processes in distributed systems. However, the underlying concepts are still the same and the designs require minor adjustments to work with threads. In this chapter, similar detector designs are discussed in relation to ours.

7.1 An Adaptable Failure Detector

Bertier et al develop an adaptable failure detector [5] similar to the one presented in this thesis. It is designed for use on a distributed system connected over a LAN, and as such each process has its own local failure detector module. A heartbeat-style detector at its core, it estimates the arrival times of heartbeats using freshness points, which are calculated three possible ways:

Chen's Estimation. The estimated arrival time is calculated using Equation 2, and a static safety margin is added to yield the freshness point.

Jacobson's Estimation. A dynamic safety margin is calculated using Jacobson's algorithm. The next freshness point is comprised of the last freshness point, the heartbeat period, and this dynamic safety margin.

Bertier et al's Dynamic Estimation. A modified, recursive version of Equation 2 and a safety margin calculated with Jacobson's algorithm comprise the freshness point.

The detector can adapt the heartbeat period during execution to meet changing requirements and account for network behavior by having all local modules come to a consensus on its new value.

Each of the estimation techniques' performance is analyzed in terms of the number of false positives and the average detection time. Dynamic estimation results in more false positives than Chen's estimation but with a significantly better detection time – 5152.6 ms versus 5672.53 ms, and 5081.49 ms for Jacobson's, each with a heartbeat interval of 5000 ms.

7.2 ϕ -Failure Detector

Hayashibara, Defago, and Katayama present a design for a failure detector without time-outs in [7]. Called the ϕ -failure detector, their design seeks to provide a detector which can adapt to both changing application requirements and changing network conditions – they have found this to be lacking, with detectors satisfying one or the other property. Their detector, which is also designed for a distributed system, does away with lists of suspected processes as well. Instead, a value ϕ_p representing the detector's confidence that the process has crashed is made accessible to the system. Querying applications can then determine independent thresholds for ϕ_p that best suit their detection needs.

An implementation of their design is proposed which would use heartbeats and their arrival time histories to determine ϕ_p . They propose to model arrival delays as a discrete probability distribution function which they smooth to yield a continuous

function. When the detector is queried, the probability of generating a false positive, P_{acc} , can be calculated using the function and the query time, and φ_p can be calculated from the probability.

7.3 Lazy Failure Detection

In [6], Fetzer, Raynal, and Tronel define a protocol for failure detection that aims to impose as little overhead as possible. Their lazy failure detection protocol monitors processes not by sending or receiving messages on a regular basis, but rather by using existing communications between processes. To support this scheme, they augment the communications primitives used by the processes with necessary failure detection data and to send acknowledgement messages back to senders. A detector implementing the protocol uses logged round trip delay data to determine if a process should be suspected. When the detector is queried, if there have been no recent communications with the process in question, a ping is sent to determine the round trip delay.

The protocol is lazy in the sense that the detector is only functioning when queried – no suspect lists are maintained, and a process’s activity is only determined in response to a query. Most other failure detectors are constantly communicating with their monitored processes and maintaining activity data, regardless of whether the information is currently needed.

Chapter 8: Conclusion

This thesis has presented the design and performance characteristics of a modular failure recovery system. Specifically tailored to the needs of the PFS_E module of CTAS, the system improves application reliability. Using a failure detector whose settings have been optimized for fast detection and few false positives, failed threads in our reimplement of the PFS_E module are swiftly detected. These failures are then repaired by the recover/repair module, which creates a new thread to take the failed one's place. Our detector's performance is slightly better than that of Bertier et al's three detectors (see Section 7.1) after normalizing with respect to polling/interrogation intervals. Overall, the entire system incurs an acceptable overhead as a result of the detector and repair mechanism.

There are several possible future directions from this thesis, the most obvious being migrating the failure recovery system to the actual PFS_E module. The failure detector may need recalibrating to determine good estimation constants, but shouldn't need design or implementation changes. While the recovery/repair module also shouldn't need many modifications, new repair procedures will need to be supplied by the PFS_E module. Since the monitored system will be more complicated, more steps may be necessary to repair threads. Furthermore, on thread failure, the repair procedure could update the thread's data, or even examine it for corruption and try to determine the cause of error.

Making the failure detector event-driven could improve overall performance. This major implementation change would require the use of a process CPU clock – the detector will never accumulate enough time to make detection effective if a thread CPU clock is used. The detector would need to maintain an ordered list of event times and set up a process CPU time alarm to signal the arrival of each timepoint. More specifically, the list would contain the timepoints for the associated events, where an event would be either the next time to send pings or a thread's freshness point. With this design, the detector would not need a main loop – the signal handler for the alarm would determine the event type, call the appropriate function to process the event, refresh the ordered list relative to the current time, and set the next alarm timepoint. The failure detector thread's CPU time should drop significantly with this modification. A similar modification could be made to the recover/repair module, repairing threads immediately rather than enqueueing them; the performance gains would be minor however.

Another possible way to improve performance is to optimize the failure detector's icode. Making the signal handler for pings even slightly more efficient will yield a huge benefit, since it's executed so often. The routine is already fairly minimal and efficient, so gains will be hard to come by.

Bibliography

- [1] Center-TRACON Automation System (CTAS) for Air Traffic Control website.
<http://www.ctas.arc.nasa.gov/>.
- [2] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, Mar. 1996.
- [3] W. Chen, S. Toueg, and M. K. Aguilera. On the Quality of Service of Failure Detectors. *IEEE Transactions on Computers*, 51(5):561-580, May 2002.
- [4] V. Jacobson and M. J. Karels. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM '88*, Stanford, CA, Aug 1988.
- [5] M. Bertier, O. Marin, P. Sens. Implementation and Performance Evaluation of an Adaptable Failure Detector. In *Proceedings of the 15th International Conference on Dependable Systems and Networks (DSN '02)*, pages 354-363, Washington, D.C., Jun 2002.
- [6] C. Fetzer, M. Raynal, and F. Tronel. An Adaptive Failure Detection Protocol. In *Proc. 8th IEEE Pacific Rim Symposium on Dependable Computing (PRDC-8)*, pages 146-153, Seoul, Korea, Dec 2001.
- [7] N. Hayashibara, X. Defago, T. Katayama. Two-ways Adaptive Failure Detection with the j-Failure Detector. In *Proc. Workshop on Adaptive Distributed Systems (WADiS)*, pp 22-27, Sorrento, Italy, Oct 2003.